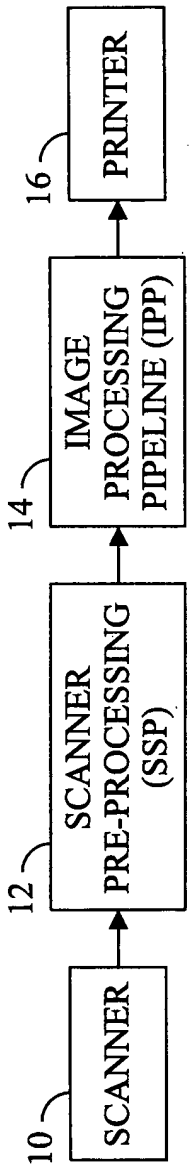

FIG. 1

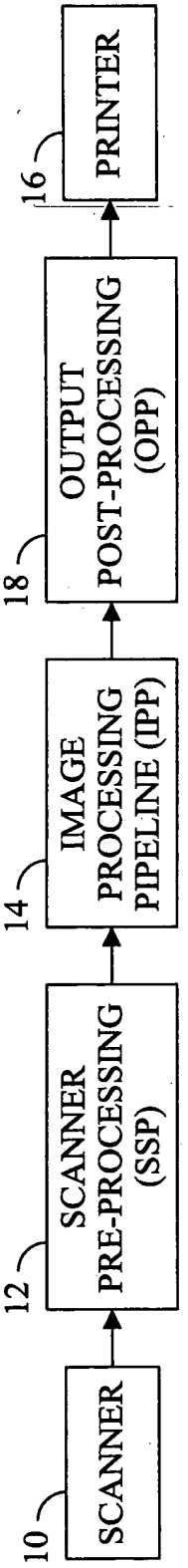

FIG. 2

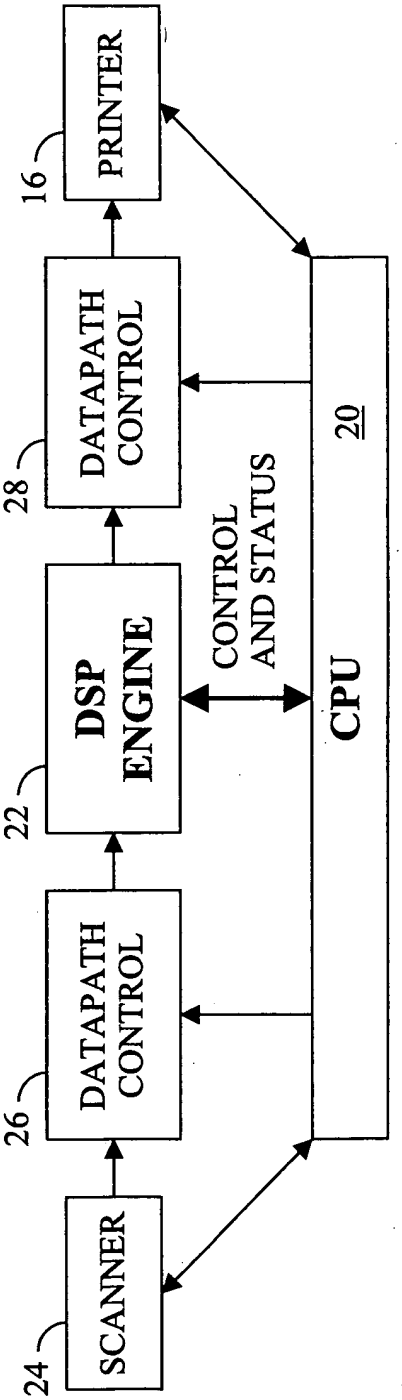

FIG. 3

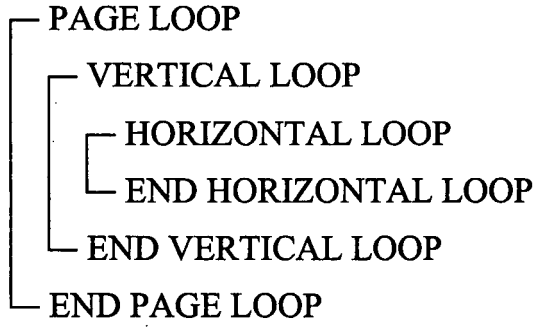

FIG. 4

```
For each output page {
  For each vertical clone {
    Scan the input document
    For each RGB scanline {
      Process the scanline with IPP
      For each resulting CMYK binary scanline {
        // Call OPP with CTP mode
        If horizontal cloning is desired
          Generate the required number of horizontal clones
        If mirroring is desired
          reverse the scanline
      }
    }
  }
}
```

FIG. 5

```
Scan the input document
For each RGB scanline {
   Process the scanline with IPP
   For each resulting CMYK binary scanline {
      // Call OPP with CTP mode
      Store each resulting scanline in memory
   }
}
For each page {
   // Call OPP with PFM mode
   For each vertical clone {
      For each CMYK binary scanline in memory {
      Retrieve scanline from memory
      If horizontal cloning is desired
         Generate the required number of horizontal clones
      If mirroring is desired
         reverse the scanline
      }
   }
}
```

FIG. 6

```
Task OPP {
   // This is where OPP's execution starts the first time it is called.
   // Initialize all necessary buffer space, variables, and registers.
L = CurrentLine = NumOPPLines;
N = CurrentVerClone = NumVertClones;
EncodedSizeCount = 0;
GlobalCompBufPointer = GlobalCompBufStart;
VertClonePadLinesToDo = 0;

TASK_LOOP:
      // The following SUSPEND causes control to return to the executive
      // (outermost) loop, which calls each of the tasks in a round-robin fashion.
      // Any or all of OPP's "local" state (registers, variables, buffers etc.) could
      // be overwritten by the executive and/or by other tasks, while OPP is suspended.
      // In the implementation, any state that must persist while OPP is suspended
      // (e.g., state flags, buffer pointers, counts, etc.) must be explicitly saved in
      // global (off-chip) memory before SUSPEND, and restored before use after
      // return from SUSPEND.  For clarity of presentation, these save/restore
      // operations are not explicitly shown in this pseudocode.

SUSPEND (&OPPReturnAddress);

      // Execution resumes here, each time (except for the very first time) that
      // the executive round-robin loop yields control back to OPP.
      // Check OPP's "idle" flag, and skip polling if cleared.
   if (!IDLE)  goto ACTIVATE_OPP;

      // Poll for OPP Activate message from CPU, and clear idle flag if received.
      // If CPU has not sent us an Activate message, then loop back to suspend
      // and give other microcode tasks a chance to run.
```

FIG. 7A

```
CheckForCPUMessage (Message, Status);
if (Status == MESSAGE_NOT_RECEIVED)  goto TASK_LOOP;
if (Message != OPP_ACTIVATE)         goto TASK_LOOP;
IDLE = 0;

ACTIVATE_OPP:
   // Check for free output buffer, unless storing.
if (DO_STORE && !DO_DECODE)  goto NO_OUTPUT_OK;
   // Loop back and suspend if insufficient output buffer space is available
   // to hold a complete CMYK line in printer format.  Note that a separate
   // output DMA task is responsible for removing data from OPP's output
   // buffer and for sending it on to the printer.
if (!BufferFree (OPPOutBuf))  goto TASK_LOOP;
   // Output buffer space is available.  If we now need to output any
   // inter-vertical-clone pad lines, then output one now, update count of
   // pad lines remaining to be output, loop back and suspend.
if (VertClonePadLinesToDo == 0)  goto NO_OUTPUT_OK;
SetToWhite (OPPOutBuf);
--VertClonePadLinesToDo;
goto OUTPUT_CMYK_LINE;

NO_OUTPUT_OK:
   // Permit decode play out even with no input.
if (DO_DECODE)  goto ENCODE_DONE;
   // Check for jull Cyan, Magenta, Yellow, and Black input buffers.
   // These would have been filled by IPP, and live in global memory.
```

FIG. 7B

```
if (!BufferFull (CyanInBuf))      goto TASK_LOOP;
if (!BufferFull (MagentaInBuf)) goto TASK_LOOP;
if (!BufferFull (YellowInBuf))   goto TASK_LOOP;
if (!BufferFull (BlackInBuf))     goto TASK_LOOP;
   // Get local copies of the Cyan, Magenta, Yellow, and Black input
   // buffers, and mark the space in global memory as empty, allowing
   // IPP to place additional data there.
GetBuf (CyanInBuf, LocalCyanBuf);
GetBuf (MagentaInBuf, LocalMagentaBuf);
GetBuf (YellowInBuf, LocalYellowBuf);
GetBuf (BlackInBuf, LocalBlackBuf);
   // Encode and measure compressed size if the mode demands it;
   // otherwise permit copy through to the printer.
if (!DO_ENCODE)  goto ENCODE_DONE;
   // Set up encoder and pointer to local compression buffer.
LocalCompBufPointer = LocalCompBufStart;
LocalSizeCount = 0;
   // Call a bitonal compression algorithm to encode each of the Cyan,
   // Magenta, Yellow, and Black inputs, one at a time, into the local
   // compression buffer.  Each Encode operation conceptually advances the
   // LocalCompBufPointer to the next free location past the thus-far-encoded
   // region, and updates the local-memory variable LocalSizeCount.
Encode (LocalCyanBuf,      &LocalCompBufPointer,  &LocalSizeCount);
Encode (LocalMagentaBuf, &LocalCompBufPointer,  &LocalSizeCount);
Encode (LocalYellowBuf,   &LocalCompBufPointer,  &LocalSizeCount);
Encode (LocalBlackBuf,     &LocalCompBufPointer,  &LocalSizeCount);
```

FIG. 7C

```
   // Update the global size-count running total, based on the local one.
EncodedSizeCount += LocalSizeCount;

   // Check running total against buffer limit.  Note: the buffer limit is sufficiently smaller
   // than the actual buffer size, to guarantee that a second scan of the same document
   // (which due to noise might compress to a slightly larger size) will also fit within the buffer.
if (EncodedSizeCount < CompressBufLimit)  goto COMPRESSED_SIZE_OK;

   // Otherwise clear encode flag, and fall back to simple CTP.
DO_ENCODE = 0;

   // If we reach here with the store flag set, it is a fatal error!  This version
   // of OPP need not be equipped to handle a buffer overflow during CTM.
   // The expectation is that CME was run successfully first, and there
   // is enough pad past the buffer limit so that the buffer can't possibly
   // overflow during the second (CTM) scan.
if (DO_STORE)  send ERROR message to CPU and ABORT;

COMPRESSED_SIZE_OK:

   // Permit play through to printer if we are not storing.
if (!DO_STORE)  goto ENCODE_DONE;

   // If we're storing, copy the valid portion of the local compression buffer.
   // out to the next free region in the global compression buffer.  Increment
   // the global compression buffer pointer by LocalSizeCount.
CopyBuf (LocalCompBufStart,  &GlobalCompBufPointer,  LocalSizeCount);

   // Disable simultaneous store and print by looping back and suspending.
   // Decrement and save input line counter until all input lines are stored.
L = --CurrentLine;
if (L)  goto TASK_LOOP;
```

FIG. 7D

```
    // Storage finished: reset line count, change state, and loop back to
    // relinquish control to the executive.  The next time we get control,
    // we will be in PFM mode.
  DO_ENCODE = 0;
  DO_STORE = 0;
  DO_DECODE = 1;
  L = NumOPPLines;
  goto TASK_LOOP;

ENCODE_DONE:

    // Check if we're decoding;  if not, play the processed data through to the printer.
  if (!DO_DECODE)  goto DECODE_DONE;

    // Bring in one line's worth of CMYK data from the global compression buffer
    // to the local compression buffer.  The compressed data format includes embedded
    // size information, so that the number of words that must be copied is determined
    // by inspection of the compressed data.
    // Set up local compression buffer pointer.
  LocalCompBufPointer = LocalCompBufStart;

    // Each copy operation transfers a line's worth of a plane's worth of image data, and
    // advances the global and local compression buffers' pointers based on the record
    // length information embedded in the compressed data.
    // copy cyan record
  GetCompressedData (&GlobalCompBufPointer,  &LocalCompBufPointer);
    // copy magenta record
  GetCompressedData (&GlobalCompBufPointer,  &LocalCompBufPointer);
    // copy yellow record
  GetCompressedData (&GlobalCompBufPointer,  &LocalCompBufPointer);
```

FIG. 7E

```
   // copy black record
GetCompressedData (&GlobalCompBufPointer, &LocalCompBufPointer);
   // Decode (decompress) a line each of C, M, Y and K from the local compression
   // buffer to the local C, M, Y and K buffers. Each Decode operation advances the local
   // compression buffer pointer to the next free location in the local compression buffer.
LocalCompBufPointer = LocalCompBufStart;
Decode (&LocalCompBufPointer, LocalCyanBuf);
Decode (&LocalCompBufPointer, LocalMagentaBuf);
Decode (&LocalCompBufPointer, LocalYellowBuf);
Decode (&LocalCompBufPointer, LocalBlackBuf);
   // Decrement and save input line counter until all input lines have been decoded.
L = -- CurrentLine;
if (L) goto DECODE_DONE;
   // We get here after the last scanline of each vertical clone. Set the number of
   // pad lines that must be output next (i.e., between the clone we just completed,
   // and the next one, if any). InterCloneVertGap is a system parameter.
VertClonePadLinesToDo = InterCloneVertGap;
   // Prepare for the next vertical clone on this page.
   // reset line count; decrement and save vertical clone counter
   // until all vertical clones have been created.
L = NumOPPLines;
N = --CurrentVertClone;
if (N) goto DECODE_DONE;
   // If we're here, we've completed a page. Reset vertical clone count and pad count.
   // and set IDLE flag so we will wait for the rest of the system components (output DMA task,
```

FIG. 7F

```
    // CPU and printer) to finish all their remaining work for this page.  Note: The output DMA
    // task maintains its own count of the number of lines per page, and it will send a message
    // to the CPU after outputting the last line of the current page.  The CPU will send an
    // Activate message to OPP after all is ready for OPP to begin playing out another page.
  N = NumVertClones;
  VertClonePadLinesToDo = 0;
  IDLE = 1;

DECODE_DONE:
#ifdef THIS_IS_A_DRAFT_PIPELINE
    // Perform 2x horizontal bit replication in place, to convert the horizontal
    // sampling rate from that of IPP (300dpi) to that of the printer (600 dpi).
  Rep2xHorizontal (LocalCyanBuf);
  Rep2xHorizontal (LocalMagentaBuf);
  Rep2xHorizontal (LocalYellowBuf);
  Rep2xHorizontal (LocalBlackBuf);
#endif
    // If the feature is selected, Perform Horizontal Cloning in place on the CMYK buffers.
  if (HorizCloningIsEnabled)  {
    HorizClone (LocalCyanBuf,     NumHorizClones);
    HorizClone (LocalMagentaBuf,  NumHorizClones);
    HorizClone (LocalYellowBuf,   NumHorizClones);
    HorizClone (LocalBlackBuf,    NumHorizClones);
  }
    // If the feature is selected, Perform Mirroring in place on the CMYK buffers.
  if (MirroringIsEnabled)  {
```

FIG. 7G

```
    Mirror(LocalCyanBuf);
    Mirror(LocalMagentaBuf);
    Mirror(LocalYellowBuf);
    Mirror(LocalBlackBuf);
  }
    // Format the CMYK data as required by the printer.
  ConvertFormat (LocalCyanBuf, LocalMagentaBuf, LocalYellowBuf,
                 LocalBlackBuf, OPPOutBuf);

  OUTPUT_CMYK_LINE:
#ifdef THIS_IS_A_DRAFT_PIPELINE
    // Send the finished image line buffer to the output DMA task, and mark the
    // buffer as full.  Output DMA task will empty the buffer when it gets a chance.
  SendBufferToOutput (OPPOutBuf);
  goto TASK_LOOP;
```

FIG. 7H